Implicit Hints: Embedding Hint Bits in Programs without ISA Changes

Hans Vandierendonck and Koen De Bosschere
Department of Electronics and Information Systems
Ghent University

***Abstract*— There is a large gap in knowledge about a program between the compiler, which can afford expensive analysis, and the processor, which by nature is constrained in the types of analysis it can perform. To increase processor performance, ISAs have been extended with hint bits to communicate some of the compiler's knowledge to the processor.**
In this paper, we propose and analyze a technique for adding or removing hints to a processor *without changing the ISA*, i.e. without breaking binary compatibility. Our technique exploits the freedom of allocating values to registers. We divide the registers in disjoint sets and assign one hint value to each set of registers.
We implement our technique in the GCC compiler. Evaluation on two very different instruction sets, the Alpha ISA and the x86-64 ISA, shows that these hints can be encoded with high accuracy, although the accuracy varies strongly between instruction sets. We demonstrate that it is possible to encode multiple hints in register names and that the quality of register allocation is not degraded.

I. Introduction

In the quest for higher performance, the boundary between compilers and architecture has faded: On one hand architectural details have been exposed to the compiler and on the other hand compilers communicate some of their analysis results to the architecture by means of hint bits. Hint bits are embedded in the instruction encoding. But instead of defining *what* the architecture must do (the semantics of the instruction) the hint bits give directions on *how* the architecture can best accomplish a task. The recent Itanium ISA [1], contains hint bits to steer branch prediction and hint bits to predict locality of memory reference.

Hint bits are a powerful tool to enhance the cooperation between compilers and architectures. Adding hint bits to an existing ISA is, however, a difficult task because of binary compatibility issues. Indeed, the instruction format must be changed to house the new hint bits but programs compiled for the old ISA must still execute correctly. Extending an ISA with hint bits may be possible a few times during the lifetime of an ISA, but not with every new processor generation.

We propose and analyze a technique to encode hints in existing instruction sets that requires no ISA changes. Our technique uses a degree of freedom in instruction encoding, namely the choice of register name, to encode hint bits. Hereby, the specification of hint bits is not part of the ISA per se and can change without breaking binary compatibility. This property makes it easier to introduce new hints in processors, but also to remove hints if they become obsolete.

A. Principle

Instruction set architectures use registers to store intermediate results. In many cases, it is legal to substitute the use of one register for another register. Due to this property, it is possible to divide the set of registers into multiple *register classes*, where each register class encodes for a particular hint. When an instruction references a register, we can use the register name to deduce a hint for this instruction.

Consider, for example, encoding a branch bias hint in conditional branch instructions (i.e. is the branch most likely taken or not-taken?). Hereto, we divide the set of registers in two register classes, e.g., the even registers to encode not-taken and the odd registers to encode taken. Using this scheme, it is hinted that the instruction `bge r1, target` is a likely taken branch while the instruction `bge r2, target` is a likely not-taken branch. Figure 1 illustrates the effect of this hint encoding when compiling the SPEC CPU2006 gcc benchmark for the Alpha ISA. The graph shows the percentage of time that conditional branches are taken (taken rate) averaged per register name. On the left, we show the taken rate when compiling gcc using an unmodified compiler. There is hardly any correlation between taken rate and register name. On the right, we see that the techniques presented in this paper force a low taken rate for even register names and a high taken rate for odd register names.

B. Contributions

We present an algorithm for encoding hints in register names. As the hints affect what registers are used by what instructions, the encoding of hints must be integrated into *register allocation*: the assignment of values (program variables, intermediate values or live ranges) to registers. We demonstrate in this paper that it is quite feasible to extend register allocation to encode hints and that the quality of register allocation is not degraded.

Furthermore, we discuss the architectural support necessary for decoding hint bits and we discuss the reconfiguration of this hardware in order to change the set of hints that are decoded.

The goal of this paper is to show the feasibility of encoding hints in register names. Hereto, we present case studies on encoding conditional branch hints and cache hit/miss hints in register names. We experimentally evaluate the accuracy of these hints on two very different instruction sets: the orthogonal RISC Alpha ISA and the CISC x86-64 ISA

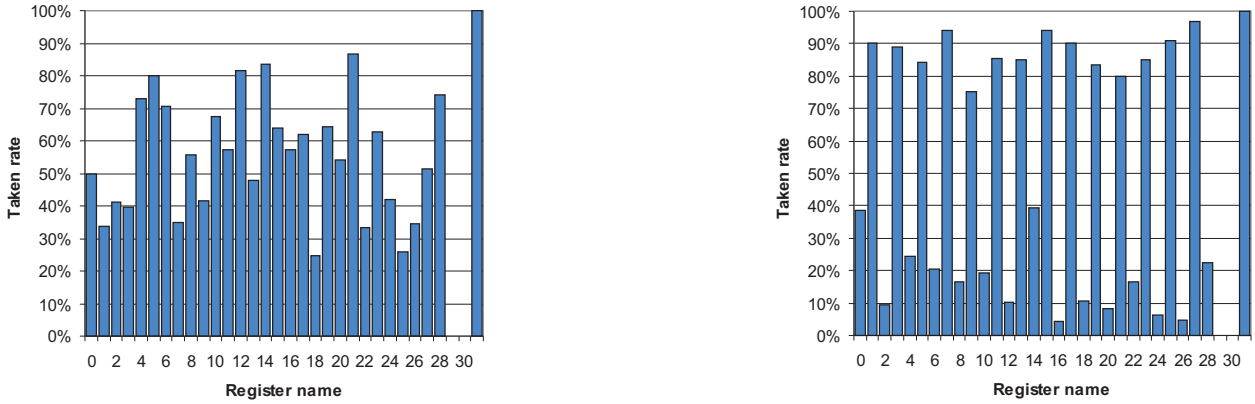


Fig. 1. Taken rate per register name in an unmodified program (left) and in a program with the branch bias encoded in the register name (right). Registers 29 and 30 have a special use and do not appear in branch instructions. Register 31 is always zero.

In future work we will consider other hints to encode in instructions and evaluate the overall speedup or energy reduction that can be achieved with these hints.

C. Paper Organization

The remainder of this paper is organized as follows. Section II cites related work on this topic. Then, we discuss the extensions to register allocation that are necessary for encoding hints in Section III. We discuss the issues related to backward compatibility in Section IV. Following, we evaluate the proposed hint encoding in Section V. Finally, we conclude the paper and provide directions for future research in Section VI.

II. Related Work

The literature provides ample evidence that there is a real practical need for encoding hints in instructions. Many processors already contain mechanisms to convey hints on, e.g., branch directions [2], [3], branch targets [4] and memory locality [5]. On the other hand, researchers have indicated that hint bits are an enabling mechanism for hardware improvements such as value prediction [6], [7], early register release [8] and criticallity-aware processors [9]. Unfortunately, there are is no generic way to actually provide these hints to the processor.

Recent ISAs have reserved the necessary bits in instructions to encode hints. It is, however, not possible to make ISA changes for every new processor generation. Thus, a technique is required that allows to encode hint bits without making ISA changes. This paper presents such a technique.

III. Implicit Hints

The compiler and the processor must both follow a set of conventions on the implicit hints. In particular, an implicit hint is specified by the following properties:

1) The instructions (opcodes) where the hint is present, e.g. a branch bias hint is only present in conditional branch instructions.
2) The operands where the hint is encoded, e.g. a hint attached to load/store instructions may be encoded in the value register operand, i.e. `r4` in `ldq r4,0(r1)`.
3) The hinted value attached to each register, e.g. taken/not-taken.

Multiple hints may be simultaneously encoded, e.g. a program may carry a branch bias hint in conditional branch instructions and a memory locality hint in load/store instructions[1]. Furthermore, the absence of a hint can be encoded by adding a register class to signify this situation.

A. Encoding Hints with the Register Allocator

Our algorithm for encoding hints in register names builds upon the notion of *register classes*. Register classes are used in many register allocators to express what registers are *interchangeable* and *independent*. Registers are interchangeable when either may be used in the same program context. Registers are independent when modifying one cannot modify the other. It turns out that commercial instruction sets do not respect the properties of interchangeability and independence [10]. To solve this problem, registers are grouped in *register classes* such that all registers in the same class are interchangeable and independent.

Figure 2 shows the register class hierarchy for a regular architecture such as the Alpha (full lines). The register class hierarchy is an acyclic graph where each node in the graph corresponds to a register class and edges denote the subset relationship. There are three register classes: the floating-point registers (FP), the integer registers (INT) and the class of all registers, which is the union of the previous two.

Additional register classes signify hints. In Figure 2, we specialize the integer registers in two classes: the registers encoding a taken branch direction (INT-T) and the registers encoding a not-taken branch direction (INT-NT). Further refinement of these register classes allows one to encode also the strength of the branch bias: whether the branch is very likely taken (strong taken) or whether it is weakly biased towards taken (weak taken). A similar refinement is performed on the not-taken branch direction.

[1] As the same register name may need to encode a combination of hints, it is advisable to divide the set of registers in different ways for the hints, e.g. by using a different bit in the binary representation of the register names to represent the hint.

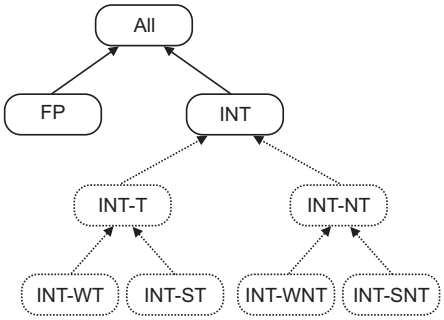


Fig. 2. Register class hierarchy extended with register classes that encode hints (drawn with dotted lines). These classes encode taken (T) or not-taken (NT) branch bias and strong (S) or weak (W) branch bias.

Multiple register classes may be legal candidates for holding a value. In such situations, it is beneficial to steer register selection by means of register class preference. When the most preferred register class has no free registers, the register allocator searches the next preferred register class for a free register, and so on. This approach is followed, amongst others, by GCC which tracks two preferred register classes per register-allocatable value.

This mechanism is used to select registers with the correct hint. E.g. the sequence of register classes INT-WNT, INT-NT, INT specifies that a value should be encoded with the "weak bias" and "taken" hints. If however all registers in the INT-WNT class are occupied, then we prefer to drop the "weak bias" hint and select a register from the larger INT-NT class. If this also fails, we select a generic INT register.

The approach outlined here has several advantages. First, we do not have to interfere with the register allocator itself as we only change the preferred register class to encode hints in register names. As such, our approach works with any register allocator. Second, it allows for a simple way to discard hints if registers encoding the correct hint are unavailable. In particular, we give preference to register classes that encode the correct hint, but when these register classes are exhausted, we fall back to the larger generic register classes. Hereby, erroneous hints may be encoded, but the value is placed in a register, which helps performance.

B. Application to Branch Bias Hints

The algorithm described above applies to most common cases of encoding hints. For branch bias hints, however, we need to apply two additional tricks.

The first problem stems from assigning branch bias hints during register allocation while the branch bias may be changed by subsequent control flow optimization and code layout. Our solution to this problem is to change the encoding of the hint: instead of directly encoding taken/not-taken information, we encode the branch bias conditionally on the branch condition. We define for every branch condition a *toggle* value and encode the following hint:

$$\text{hint} = \text{branch bias} \quad \text{XOR} \quad \text{toggle(branch condition)}$$

where the taken branch bias is represented by 1 and the not-taken branch bias is represented by 0. The power of this encoding is in the choice of the toggles: when a branch bias is reversed, then the condition toggle reverses too. A possible assignment of toggles is listed in Table I.

TABLE I
BRANCH CONDITION TOGGLES.

Condition	Toggle	Condition	Toggle
equal	0	not-equal	1
greater-than	0	less-or-equal	1
greater-or-equal	0	less-than	1

The second problem concerns instructions where all used registers are implicitly named, e.g., the flags or condition code register in many ISAs. It is not possible to directly encode hints in the register arguments. Instead, we select another instruction to act as a proxy for encoding the hint. We select the instruction that produces the implicitly named register. Consider the following sequence of x86 instructions:

```
cmp %rax,%rdx     # compare, proxy inst.
bne target        # branch if not equal
```

The `cmp` instruction allows choice in the register names of its operands. We encode the hint in its first register operand.

IV. Decoding Implicit Hints

A compiled program where hints are encoded in register names is not directly distinguishable from a program without hints, as both use the same ISA. It is therefore necessary to inform the processor about the presence of hints. The processor can then look for these hints as it decodes the instructions.

A. Decoding Mechanism

The processor decodes hints during instruction decode. Depending on the opcode and on the types of register operands, it determines what register operand encodes a hint and it extracts the hint from the register name. This process is very simple and can be implemented by means of a combinatorial circuit.

Figure 3 shows the hint decoding hardware. A hint configuration register determines what hint may be encoded in register names. Based on this information, one can deduce from the opcode if the hint is present in this instruction and one can extract the hint from register names (bottom). Extracting the hint requires the opcode in order to correctly re-apply branch condition toggles (Section III-B).

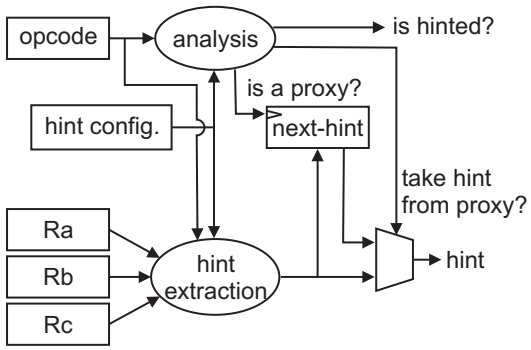


Fig. 3. Decoding a hint in a proxy instruction.

When hints are encoded in instructions that read only implicitly named registers, e.g. as in branch instructions that check a flags register, then we encode the hint in a proxy instruction (Section III-B). This makes the decoding process a little more complicated, as the hint must be decoded from the proxy instruction and remembered until the dependent instruction is encountered. Hereto, we track dependencies on the flags register. When decoding an instruction that writes the flags register, the decoder logic generates a signal to store the hint in the "next-hint" register. When decoding an instruction that reads the implicit register, the hint is picked-up from the next-hint register.

B. Configurable Hints

Figure 3 contains a user-programmable *hint configuration* register to specify what hints have been encoded in the currently executing code. Only the hints indicated in the hint configuration register are actually decoded and used by the processor. This register adds a level of *configurability* to the presence of hints. The hint configuration register is however architecturally visible.

Configurable hints have some useful applications. First, making hints configurable allows a smooth transition path for adding hints to a processor and for removing them. When a processor implements a new hint, then programs compiled for older processors have the new hint turned off by default. The hint will be encoded in freshly compiled programs, where the hint is turned on. Furthermore, hints may become redundant in which case it is straightforward to reclaim the hint bits and to implement a different hint. This is all possible without bloating the ISA and without breaking binary compatibility.

Second, it allows one to encode only those hints that are most useful for a particular code section. E.g. it may be more beneficial to encode memory locality hints in memory-dominated applications than it is to encode branch hints. Third, by making the encoded hints selectable, it is also possible for the processor to implement a portfolio of hints that is significantly larger than the small number of hints that can be encoded in a typical register name space. Furthermore, a processor can implement hints targeting different goals, e.g. performance, power consumption, etc.

C. Backwards Compatibility

Processors that recognize implicit hints must still perform well when running code where no hints are encoded in register names. Also, there is a potential problem when linking (either static or dynamically) object files where some files have implicit hints and others do not.

To handle these scenarios, we rely on the hint configuration register to enable or disable hints during execution. For instance, it allows programs without hints to run while the hardware does not attempt to read implicit hints. Hereto, system software turns off implicit hints using the hint configuration register when the program is loaded into memory.

Linking object files with and without implicit hints is possible by inserting code snippets to set the hint configuration register at module boundaries. E.g. when calling a function from a different module, the call instruction jumps to a preamble of the function that sets the hint configuration register. When returning from a cross-module call, the hint configuration register is reset to the appropriate value. Within a compilation module, the compiler has absolute knowledge of the value of the hint configuration register. This is much the same mechanism as used on Alpha systems for keeping the global pointer ($gp) consistent. It guarantees that, whatever the control flow in the program, the hint configuration register (or the global pointer) contains the desired value.

TABLE II

THE RELATION BETWEEN BRANCH HINTS AND REGISTER NAMES. IN X86-64, SUBREGISTERS (E.G. AL, AH, AX AND EAX) ENCODE THE SAME HINT AS THE 64-BIT REGISTER (E.G. RAX).

Hint value		ISA	
Bias	Strength	x86-64	Alpha
not-taken	weak	rax, rdx, r8, r10, xmm{0,2,8,10}	R*n*, F*n* if $n \bmod 4 = 0$
not-taken	strong	rsi, rbp, r12, r14, xmm{4,6,12,14}	R*n*, F*n* if $n \bmod 4 = 2$
taken	weak	rbx, rcx, r9, r11, xmm{1,3,9,11}	R*n*, F*n* if $n \bmod 4 = 1$
taken	strong	rdi, rsp, r13, r15, xmm{5,7,13,15}	R*n*, F*n* if $n \bmod 4 = 3$

V. EVALUATION

The goal of this evaluation is to confirm the feasibility of encoding hints in register names and to gauge the impact of ISA properties. Speed-up results are deferred to future work.

A. Evaluation Environment

We experimentally evaluate the accuracy of implicit hints on the RISC Alpha ISA and the CISC x86-64 ISA using branch bias and branch strength hints.

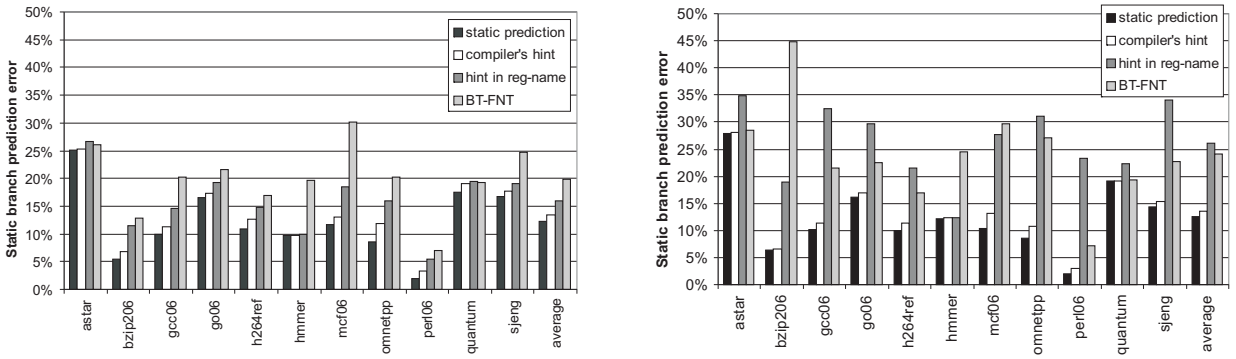
The relation between branch hints and register names is depicted in Table II. Branch hints are encoded in the register argument of conditional branches in the Alpha. In the x86-64, the hints are encoded in a proxy instruction.

We implemented the hint encoding techniques in a non-research compiler: GCC 4.2.0. We modified the Alpha and x86-64 back-ends to incorporate the new register classes. For the x86-64, we broke up the GENERAL (integer registers), INDEX (pointer registers) and SSE (floating-point/SSE2) register classes to encode hints. For the Alpha, the integer and floating-point register classes were refined.

The accuracy of the encoded hints is computed using functional simulation. We use SimpleScalar for the Alpha [11] and PTLsim for the x86-64 [12]. Only the application part of the benchmarks is measured. Libraries are not modified and are not included in the reported statistics. All benchmarks are run to completion.

B. Branch Hints

We evaluate the accuracy of encoding a branch bias hint by the overall static branch prediction accuracy, i.e. the number of times that the branch bias hint corresponds to the actual branch direction during execution. Figure 4 shows



(a) Alpha (b) x86-64

Fig. 4. Prediction accuracy for the branch bias hint in the Alpha and x86-64 broken down by cause of prediction error. The "static prediction" bar shows the oracle static prediction accuracy. The "compiler" bar shows the accuracy of the hints in the internal representation of the compiler. The "hint in reg-name" bar adds the impact of register pressure and ISA constraints to the accuracy. The "BT-FNT" bar compares to the accuracy of backward-taken forward-not taken static branch prediction.

the prediction accuracy of the hint bit during various steps in the compilation chain.

The perfect static branch bias predictor sets an upper bound on the achievable static branch prediction accuracy. Prediction errors occur as branches that go the same direction D% of the time are statically mispredicted in at least $min(D\%, 100\% - D\%)$ of the cases. This variability of branch direction introduces a minimum error of 12.5% in both ISAs.

The next source of error is the compiler. Compiler hints may be wrong as it can be inherently impossible to correctly compute branch biases when transforming a control flow graph [13]. E.g. for code duplication, the compiler must guess the branch biases in each copy of the code. An obvious assumption is that all copies of the code behave the same as the original code, but this is not true in general. The compiler's branch bias is slightly distorted resulting in less than 1.3% average increase of prediction error.

Encoding the compiler's erroneous branch bias in register names further increases prediction error to 15.91% in the Alpha, which introduces an error of 3.72% compared to the perfect static branch predictor. For the x86-64, hint encoding is far less accurate due to various ISA properties, such as abundant use of implicitly named registers and 2-operand instruction formats allowing memory operands.

The difference between the hint in reg-name bar and the compiler's hint bar shows hint encoding errors. This includes errors due to register class restrictions and register pressure but also due to ISA properties and due to incompleteness of the implementation of the technique (rare cases are not always implemented).

The backward taken/forward not-taken (BTFNT) static branch predictor is a well-studied static predictor that assumes that backward branches are typically taken as they often correspond to loop branches, while forward branches are generally not-taken. The branch prediction error for BTFNT is 19.88% on average on the Alpha and 24.09% in the x86-64. Thus, on regular ISAs, encoding hints in register names is more accurate than simple static hardware predictors.

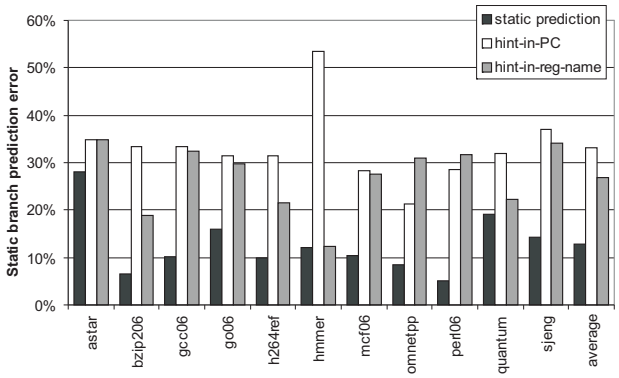


Fig. 5. Analysis of encoding the branch bias hint in the program counter on the x86-64 ISA.

C. Encoding Hints in the PC

Jiménez [14] proposes to encode branch hints in the program counter. By carefully aligning the code, it is possible to make a particular bit in the program counter collide with the branch bias. The code is aligned by inserting no-ops, which are placed as much as possible on cold code paths. Overall, placement is not exact. We refer to [14] for details of the algorithm.

Figure 5 demonstrates the error on the branch bias hint when it is encoded in the program counter or in register names. Results are shown for the x86-64 ISA because Jiménez also applied his work to x86-64. The hint is on average 6% more accurate when it is encoded in the register name then when it is encoded in the program counter. Thus, register names are a more appropriate place to encode hints, even though the x86-64 ISA is strongly register-constrained. In the case of the hmmer benchmark, the structure of the code does not provide many degrees of freedom for aligning the code. Furthermore, strong deviations between training and reference inputs push the error above 50%.

D. Multiple Hints

We have set up an experiment where 2 hints are encoded: a branch bias hint and a branch strength hint, indicating that

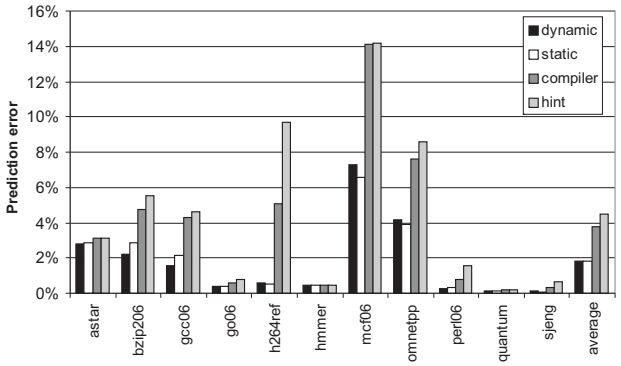


Fig. 6. Predictability of cache hit/miss property of load instructions in the Alpha ISA. A comparison is made to the dynamic prediction accuracy with a hit/miss predictor similar to the Alpha 21264's predictor.

a branch is strongly biased (when the bias is larger than 95% or less than 5%) or it is weakly biased. Prediction error of the bias hint is increased with 0.27% points.

E. The Cost of Implicit Hints

The overhead of encoding hints in register names is practically zero: We measured the dynamic instruction count increase for encoding a branch bias hint and a branch strength hint, averaging at 0.15% and 0.27% in the Alpha and x86-64 ISAs respectively. The numbers vary across benchmarks between -0.38% and 0.75% for the Alpha and between -0.05% and 0.56% for the x86-64. This shows that the goal of minimizing spill code, that we set forth at the beginning, has been successfully reached.

F. Cache Hit/Miss Hints

Implicit hints are not restricted to branch hints but they are useful for a variety of different hints. Here, we discuss the cache hit/miss hint. We simulate accesses to a 64 KB, 4-way set-associative data cache with 64-byte blocks. For each load instruction in the program, we determine what is most likely: a hit or a miss. This information is encoded in the register name.

The cache hit/miss hint is, on average, wrong 4.49% of the time (Figure 6). We find that most of the errors in the static prediction are due to the compiler, as the hint that the compiler tries to encode has an error rate of 3.72%. These errors are due to the absence of profiling information for many load instructions.

The graph also shows the accuracy of a dynamic hit/miss predictor (e.g. a table of 4-bit bimodal counters), which has an error rate of 1.8%. It is clear that the dynamic predictor has a smaller error. The point is however that comparable results can be obtained using static information and that using static information has many advantages in terms of hardware cost, energy consumption, etc.

VI. Conclusion and Future Work

This paper proposes implicit hints, a technique to embed hints in programs *without changing the ISA*. This technique exploits the freedom present in the choice of register names to hold information. We have presented a compiler algorithm for encoding implicit hints and we have discussed the hardware support necessary for decoding those hints.

Evaluation on the Alpha and the x86-64 instruction sets has shown that hints can be encoded with high accuracy, although the accuracy depends strongly on the ISA. Branch bias hints are encoded with 3.72% error in the Alpha and with 14.22% error in the x86-64. We showed that encoding an additional hint in register names does not significantly degrade this accuracy and that encoding hints in register names does not affect the quality of register allocation.

Acknowledgments

Hans Vandierendonck is a Postdoctoral Fellow with the Fund for Scientific Research–Flanders. This research was also sponsored by Ghent University and by the European Network of Excellence on High-Performance Embedded Architectures and Compilation.

References

[1] "Intel Itanium architecture software developer's manual," Jan. 2006, document Number: 245319-005.

[2] *PowerPC User Instruction Set Architecture. Book I*, IBM, Sept. 2003, version 2.01.

[3] *Intel Pentium 4 Processor Optimization: Reference Manual*, Intel Corporation, 1999, order Number: 248966-001.

[4] *Synergistic Processor Unit Instruction Set Architecture*, IBM, Jan. 2007, version 1.2.

[5] H. Sharangpanai and K. Arora, "Itanium processor microarchitecture," *IEEE Micro 2000*, vol. 20, no. 5, pp. 24–43, 2000.

[6] M. Burtscher, A. Diwan, and M. Hauswirth, "Static load classification for improving the value predictability of data-cache misses," in *Proceedings of the ACM SIGPLAN'2002 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2002, pp. 222–233.

[7] Q. Zhao and D. J. Lilja, "Static classification of value predictability using compiler hints," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 929–944, Aug. 2004.

[8] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin, "Compiler directed early register release," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 110–122.

[9] P. Salverda, C. Tucker, and C. Zilles, "Accurate critical path prediction via random trace construction," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 64–73.

[10] M. D. Smith, N. Ramsey, and G. Holloway, "A generalized algorithm for graph-coloring register allocation," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004, pp. 277–288.

[11] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 36, no. 2, pp. 59–67, Feb. 2003.

[12] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2007)*, Apr. 2007, pp. 23–24.

[13] Y. Wu, "Accuracy of profile maintenance in optimizing compilers," in *Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2002.

[14] D. Jiménez, "Code placement for improving dynamic branch prediction accuracy," in *Proceedings of the ACM SIGPLAN'2005 Conference on Programming Language Design and Implementation*, 2005, pp. 107–116.